Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5540--07-9067

Xenon Formal Security Policy Model

JOHN MCDERMOTT
JAMES KIRBY
MYONG KANG
BRUCE MONTROSE

Center for High Assurance Computer Systems
Information Technology Division

August 14, 2007

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
14-08-2007	Final Report	Jan 2007 – Apr 2007

4. TITLE AND SUBTITLE

Xenon Formal Security Policy Model

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

John McDermott, James Kirby, Myong Kang, and Bruce Montrose

5d. PROJECT NUMBER

5e. TASK NUMBER
IT-235-007

5f. WORK UNIT NUMBER
6475

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory, Code 5540
4555 Overlook Avenue, SW
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/5540--07-9067

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR / MONITOR'S ACRONYM(S)

11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Xenon is a high-assurance virtual machine monitor based on the Xen open-source hypervisor. We model the Xenon high-assurance virtual machine monitor's security policy as a conditional non-interference policy, using an event-based paradigm and the Communicating Sequential Processes (CSP) formalism. Our single model formally describes not only the separation of information flow but also the sharing. We also present our strategy for showing correspondence between this model and the Xenon interface.

15. SUBJECT TERMS

Formal security policy model
Virtual machine monitor
CSP
Non-interference

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			John McDermott
Unclassified	Unclassified	Unclassified	UL	28	**19b. TELEPHONE NUMBER** *(include area code)* (202) 404-8301

CONTENTS

1 Introduction 4

1.1 Conditional MSL Information Flow Policy 2
1.2 The MSL Separation Policy 3
1.3 The MSL Sharing Policy 4
1.4 The CSP Formalism 4

2 The Formal Model 5

2.1 Interfaces 6
2.2 Events and Data Values 6
2.3 Guest OS Processes 7
2.4 Boundary Controller 9
2.5 VMM 10
2.6 Complete Model 12

3 Rationale 13

3.1 Deadlock Freedom 14
3.2 Separation 14
3.3 Sharing 17

4 Semiformal Correspondence Demonstration 18

4.1 Basic Mapping 19
4.2 Externally Visible Data 20
4.3 Request-Response and Transients 21
4.4 Nondeterministic Interrupts 21
4.5 Subjects and Objects 22
4.6 Least Privilege 22
4.7 Using the Model and Correspondence Demonstration 23

5 Summary 23

References 23

Xenon Formal Security Policy Model

J. McDermott, J. Kirby, M. Kang, and B. Montrose
Naval Research Laboratory

14 June 2007

Abstract

Xenon is a high-assurance virtual machine monitor based on the Xen open-source hypervisor. We model the Xenon high-assurance virtual machine monitor's security policy as a conditional non-interference policy, using an event-based paradigm and the CSP formalism. Our single model formally describes not only the separation of information flow but also the sharing. We also present our strategy for showing correspondence between this model and the Xenon interface.

1 Introduction

This report defines the formal security policy model for the Xenon high-assurance virtual machine monitor (VMM) version of the Xen hypervisor [1, 2]. High-assurance virtual machine monitors are used to provide highly robust information flow control, tamper-resistance, and self-protection for communities of users, based on shared interests or responsibilities of those communities. The distinction that protection is not being provided for individual applications, network protocols, or users. Instead, host processing for all applications, network protocols, and users hosted by an execution environment is separated on the basis of information flow control between user communities. For example, suppose a large business forms a new partnership with another business that is also a potential competitor. The partnership involves joint research and development of a new product line. That partnership constitutes an interest group or community for the employees working on the partnership. The community of interest working on this new product line needs a highly robust shared execution environment that controls information flow. Protecting and separating network communication for this community is necessary but not sufficient, since weak execution environments at the end points of communication can be compromised.

Information flow control in execution environments is a difficult problem. In contrast to communication networks, execution environments provide processing as well as storage and communications. Because of this, they have a much richer set of possible operations. Interactions among this richer set of operations can cause many different forms of unintentional information flow, as a side-effect of intentional or authorized processing. Strong

Manuscript approved June 27, 2007.

information flow control must address all of these possible forms of information flow. Virtual machine monitors finesse the problem by partitioning these large sets of possible flows into relatively small numbers of virtual execution environments, based on user communities. Some virtual machine monitor implementations are candidates for strong information flow security because their implementations are small and do not evolve rapidly [3].

The Xenon formal security policy model serves as: 1) a key element in the policy-to-code modeling chain, and 2) evidence for a Common Criteria security evaluation. A formal security policy model is needed to meet the Common Criteria requirements for high-assurance. Xenon currently considers high assurance to approximate the Common Criteria EAL6 package, depending on which version [4] of the Common Criteria are used.

1.1 Conditional MSL Information Flow Policy

Xenon's basic approach to separating user communities is a *multiple single levels (MSL)* information flow policy:

- each community is given a separate execution environment and
- information flow between the environments is limited to a restricted form of replication, i.e. when an execution environment wants to read data from another domain, it reads a local replica.

For Xenon, the MSL separation policy is formally modeled as a special kind of conditional non-interference policy [5]. A non-interference policy is defined for two sets of subjects *High* and *Low*: information is not allowed to flow from *High* to *Low*. Non-interference policies define the way in which a *High* subject is prevented from interfering with the behavior of a *Low* subject. In terms of the security domain lattice, case 2 has domain dom_i as *Low* and dom_j as *High* (dom_j may not interfere with dom_i), and case 3 has both domains dom_i and dom_j in the role of *High* (neither domain may interfere with the other). A conditional non-interference policy allows some flows from *High* to *Low*, via restricted communication paths.

The Xenon MSL policy begins with a definition of the security domains that serve as sources and destinations of information flows. We take security domain names (e.g. i and j) from some finite set of the nonnegative integers $D = \{0, 1, \ldots, b-1\}$. (Later in the model we also use singleton set $\{b\}$ for the top-level system domain that is always authorized to see anything on the system. Then our domain names are taken from $D \cup \{b\}$.)

For any pair of security domains (dom_i, dom_j), one of three policy relationships exists:

1. information may flow in both directions between the domains (we say $dom_i = dom_j$);
2. information may flow from dom_i to dom_j (we say $dom_i \rightsquigarrow dom_j$);
3. information may not flow between dom_i and dom_j (we say that the security domains are incomparable).

If we assume least upper and greatest lower bound security domains then this is of course the classical information flow lattice of Denning [6, 7]. In the Xenon formal model we call this lattice the *security domain lattice*. It is necessary to talk about the security domain lattice because some flows allowed by the MSL information flow policy do not obey the security domain lattice relationship.

The MSL information flow policy is a two-part policy defined over the security domains: a *separation policy* and a *sharing policy*. The separation policy prohibits all information flows between domains. The sharing policy modifies this to allow only flows between a pair of domains when one of the domains is the *boundary controller* domain that acts as the least upper bound of all the security domains. Restated, the sharing policy only allows direct flows to or from the boundary controller domain. All other flow is prohibited. The sharing policy also restricts the flows allowed from the boundary controller itself. Intuitively, the flows allowed by the boundary controller either replicate information "up" or downgrade information back to a "lower" security domain.

1.2 The MSL Separation Policy

There is a large body of mathematical work on confidentiality, information flow, and separation policies. This work had its beginnings in Bell-LaPadula [8] and Denning [6,7]. Goguen and Meseguer's *non-interference* [5] represents an important mathematical refinement that led to extensive research on information flow policies. The work of McLean [9] unified many of these various models of information flow security. Shortly after this, process algebra researchers made significant progress in two areas: expressing formal non-interference models in a widely understood general purpose notation and clarifying some of the more subtle issues related to McLean's model. Significant examples of this work are Roscoe et al.'s concepts of lazy and eager abstraction, [10], Roscoe's use of determinism in non-interference policy modeling [11] and Ryan and Schneider's process algebra generalization of non-interference [12].

Recent work in security modeling [13] has focused on the safety problem for security [14]. Since this problem is focused on access control policies rather than information flow policies, there is little application to our Xenon model.

Recent work on MILS separation kernels has produced a new kind of formal separation policy model written directly in machine processable ACL2 [15, 16]. The key difference in this formal model is not that it uses a machine processable notation but that it models separation in terms of memory segments, i.e. *infiltration* and *exfiltration*. Defining separation in terms of memory segments instead of events could be problematic for behavior that could be made to change without violating segment rules. Examples would be inter-process communication primitives such as locks or semaphores, machine instruction results, hypercall results, traps, and interrupts. For an informal example, suppose a VMM implementation has a hypercall h that returns an integer value into a segment that the guest requesting the hypercall can read. A VMM that was not secure might allow a second guest to cause the VMM to return $1 = h(x)$ to the requesting guest some of the time and $0 = h(x)$ at other times. Modulating hypercall, instruction, or interrupt results like this would not violate any

policy regarding the state of memory segments. Further discussion of this issue is outside the scope of the Xenon work. We did not try to address this in Xenon but chose to extend the concept of more abstract policy modeling that began with Bell-LaPadula.

1.3 The MSL Sharing Policy

Usable systems support controlled sharing of information across domains that are otherwise to be separated. The MSL formulation of this policy breaks the allowable information flow into two parts: 1) flow in the direction defined by the security domain lattice, when $dom_i \rightsquigarrow dom_j$ (intuitively from *Low* to *High*), and 2) flow against the direction defined by the security domain lattice (downgrading).

1.4 The CSP Formalism

The Xenon formal security policy model is written in CSP [17, 18] and is based on the work of Roscoe in defining non-interference policies via nondeterminism [11]. Process algebras like CSP are well-suited to modeling non-interference and other information flow security policies because they are trace-based and ultimately information flow security is defined over sets of system traces [9]. This subsection presents a brief informal review of CSP as used in the security policy model, to assist the reader in understanding the model. Following the proofs will require more knowledge of CSP. The specific CSP syntax of the model follows Roscoe [17]. Unless otherwise specified, the model uses stable failures denotational semantics [17, 18].

CSP is a calculus for reasoning about patterns of communication between multiple threads of computation. Communication is instantaneous; ordinary CSP does not model the duration of events. The communication patterns are defined as sets of traces of events. Events are instantaneous but may be compound, in the sense that instantaneous event $x.i.j$ has three values that are communicated to any process that shares that event. The direction of communication is irrelevant, although CSP provides syntactic sugar to remind the reader which process is meant to be the sender or receiver. For example, in the send event $hyp!op$ the "!" replaces the dot separator "." in $hyp.op$, indicating a value op sent on a communication channel named hyp. The same event written $hyp?op$ would remind the reader that the event is being interpreted as a receive, in the process containing it.

The fundamental CSP construction is the *process*, an entity that generates or participates in events chosen from its *alphabet*. A (recursively defined) sequential process P_1 that generates traces such as $\langle a, a, a \rangle$ is written as $P_1 = a \rightarrow P_1$. The set of all possible traces of a process P_1 is denoted $\mathsf{traces}(P_1)$. The traces of process P_1 include the empty trace $\langle\rangle$, as well as $\langle a \rangle$, $\langle a, a \rangle$, and the unbounded trace $\langle a, a, a, \ldots \rangle$. The process $P_2 = a \rightarrow b \rightarrow P_2$ generates traces of the form $\langle a, b, a, b \ldots \rangle$ with a and b events in strict alternation. For terminating processes there are constant processes like $SKIP$ that do nothing. A sequential process communicates with its environment whenever it agrees with its environment about the next event that can happen. If a process has no possible agreement with its environment then it is equivalent to the constant (deadlocked) process $STOP$. A sequential

process can offer an initial choice of (component) processes: $P_3 = c \rightarrow P_3 \,\Box\, d \rightarrow P_3$ has the trace $\langle d, d, d, d, c, d, c, c, d\rangle \in \mathsf{traces}(P_3)$, in its set of traces. The *external choice operator* $\Box$ shows that process P_3 begins with a choice, made by P_3's environment, of either the left hand component $c \rightarrow P_3$ or the right hand component $d \rightarrow P_3$. Process $Q_1 = a \rightarrow Q_1 \,\Box\, b \rightarrow \mathit{SKIP}$ can never have more than one b event in its traces and no a events may follow the b event; if Q_1's environment ever chooses b then it gets no more chances to choose the left hand component. CSP choice operators can be indexed, just as addition $+$ can be indexed to become summation Σ. An example of indexed external choice from the formal model is the process $GSHR_i$ in Equation 3 where external choice is indexed over possible downgrading requests $x.i.j$.

Sequential processes model a single thread; we can apply *concurrency operators* to groups of sequential processes to model multi-threaded computation. Processes in communication that agree on the next event will share that event. The easiest concurrency operator to understand has no sharing at all: *interleaving* $|||$ which combines its operands to execute concurrently without communication. So process $Q_2 = P_1 \;|||\; P_2$ can have traces $\langle a, a, a, b, a, b, a\rangle$ and $\langle a, b, a, b, a, b, a\rangle$. Generally, it is nondeterministic as to which process is responsible for identical events when they are concurrent but not communicated. Without communication, the identical event occurs separately in each process. Since our model needs determinism at certain points, we avoid interleaving over processes with identical events. Interleaving can be indexed as well; Equation 3 has process G_i defined by indexed interleaving.

For concurrency with communication, our model uses the *alphabetized parallel* operator $\underset{X}{\|}_{Y}$ with interfaces X and Y. The interfaces of alphabetized parallel are event sets that show which events are communicated on the corresponding side of the parallel operator. Events that are in $X \cup Y$ are synchronized over the two processes. So if we define interfaces $X = Y = \{a\}$, then process $Q_3 = P_1 \underset{X}{\|}_{Y} P_2$ has the same traces as process P_2 because both P_1 and P_2 execute their a events at the same time.

A final bit of CSP used in the model is a *parameterized process*, as in parameterized process $P(x)$. In this process, parameter x is either an event or part of an event that is used in the definition of process $P(x)$, e.g. $P(x) = x.i.j \rightarrow P(x)$.

2 The Formal Model

Our formal model of the Xenon security policy uses a specific notion of separation called *independence* [17] that defines separation between *Low* and *High* as the independence of *Low*'s view from anything *High* might do.

If S represents the VMM and its guest operating systems as a (composite) CSP process, H is the set of *High* CSP events, and L is the set of *Low* CSP events, then the separation policy will have the form

$$\mathcal{L}_H(S) = (S \underset{X}{\|}_{H} \mathit{CHAOS}_H) \setminus H \;\mathsf{det} \tag{1}$$

where the equivalence is stable failures equivalence and the notation P det means process P is deterministic. Following Roscoe [17], a process P is deterministic (P det) when

$$s^\frown\langle a\rangle \in traces(P) \implies (s, \{a\}) \notin failures(P) \tag{2}$$

The best explanation of Equation 1's definition can be found in Chapter 12 of Roscoe's text [17]. This definition "subsumes" the *High* user (guest operating system) with the CSP constant $CHAOS_H$ that not only can perform all possible sequences of *High* events, i.e. sequences a well-behaved user would not generate, but also can arbitrarily refuse to perform any of them as well.

Intuitively, Equation 1 says that if a malicious guest operating system acting without any restraint can cause some non-determinism in *Low* behavior (i.e. there aren't any system or *Low* events that can restrict that particular behavior) then that malicious guest operating system can leak information against the allowable direction of information flow.

The introductory formulation of Equation 1 omits internal details about the system S that is supposed to satisfy the MSL separation policy. These internal details include the security domains, the guest operating systems, the boundary controller, and the VMM itself. We now present a complete model that adds all of these details to the process S. We begin with the individual events and channels used to communicate between processes, then we describe the component processes that define communication patterns, and finish with a presentation of the combined system.

2.1 Interfaces

The complete model uses four sets of communication channels between its interfaces. The first two sets contain channels used for the separation policy

- $hyp_i, i \in D \cup \{b\}$, that communicate events initiated by the guest corresponding to security domain i and,
- $sig_i, i \in D \cup \{b\}$, for events initiated by the VMM, for security domain i.

Figure 1 depicts the use of the remaining two sets of channels, for sharing. Channels $rl_i$and rr of the *request channel* set are used to communicate requests to share data across domains, to and from the boundary controller. Channels sl and sr_i of the *share channel* set are used by the boundary controller and VMM to forward authorized sharing.

2.2 Events and Data Values

Each event has a *security class* that is the same as the security domain of the highest domain participating in the event. The VMM process shares every event with at least one guest process G_i but has no security domain itself. There are three kinds of events:

- separation policy events $hyp_i.op,\ sig_i.e \in SEP$ that model the hypercalls and signals between guest processes and the VMM,

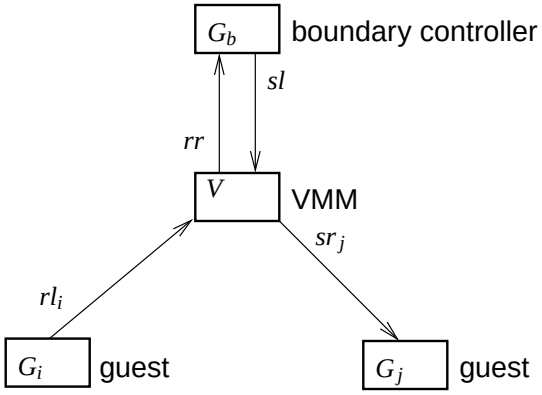


Figure 1: Communication Channels Between Process Interfaces

- sharing request events $rl_i.x.i.j, rr.x.i.j \in REQ$ that request sharing of data x from domain i with domain j, and
- sharing policy events $sl.x.i.j, sr_i.x.j.i \in SHR$ that transmit shared data x to domain j.

So the sets SEP, REQ, SHR contain the separation, request, and share policy events respectively. These events are used to describe the allowed sequences of interaction between the VMM, its guests, and the boundary controller. We also partition these sets on a security domain basis, e.g. SEP_i is the set of separation policy events for security domain i.

The values $x.i.j$ associated with events in set REQ are taken from two sets YES and NO that model the rules used by the boundary controller to restrict sharing of data across security domains. Set YES contains values of the form $x.i.j$ where x is the value to be passed across a security domain and i and j are the source and destination of the flow, and the sharing is authorized[1]. The set NO contains all other possible sharing arrangements that could be requested, i.e. unauthorized sharing; it is the complement of YES with respect to model events of the form $x.i.j$.

The use of the ! and ? separators in the model has no semantic significance but merely serves as a reminder about which process is "sending" (!) or "receiving" (?) during a synchronized event. When send or receive does not matter, we may use a dot (.) to replace the ! or ? separator.

2.3 Guest OS Processes

The guest operating systems are modeled by a finite set $\{G_k \mid k \in D\} \cup \{G_b\}$ of guest operating system processes G_i. Figure 2 shows GSPML diagrams [19, 20] of the guest process structure. The boundary controller guest operating system process G_b runs in the

[1]The set YES contains not only events representing flows in the direction defined by the security lattice but also flows against that direction, that the users have authorized.

boundary controller security domain, i.e. it is not part of the VMM but in domain b. A guest operating system process may also be referred to as a guest process or just a guest. The guests run interleaved as $G = \;|||\, G_i, \quad i \in D \cup \{b\}$. Each guest process $G_i, i \neq b$, is defined as the interleaving of a separation component and a sharing component

$$G_i = GSEP_i \;|||\; GSHR_i \tag{3}$$

where

$$GSEP_i = hyp_i!op \rightarrow GSEP_i \,\Box\, sig_i?e \rightarrow GSEP_i$$

and

$$\begin{aligned} GSHR_i = \\ & \Box_{x.i.j \in YES}\; rl_i!x.i.j \rightarrow GSHR_i \\ & \Box_{x.i.j \in NO}\; rl_i!x.i.j \rightarrow GSHR_i \\ & \Box\; sr_i?x.j.i : YES \cup NO \rightarrow GSHR_i \end{aligned}$$

Interleaving is used in the definition of the guests to isolate the separation behavior from the

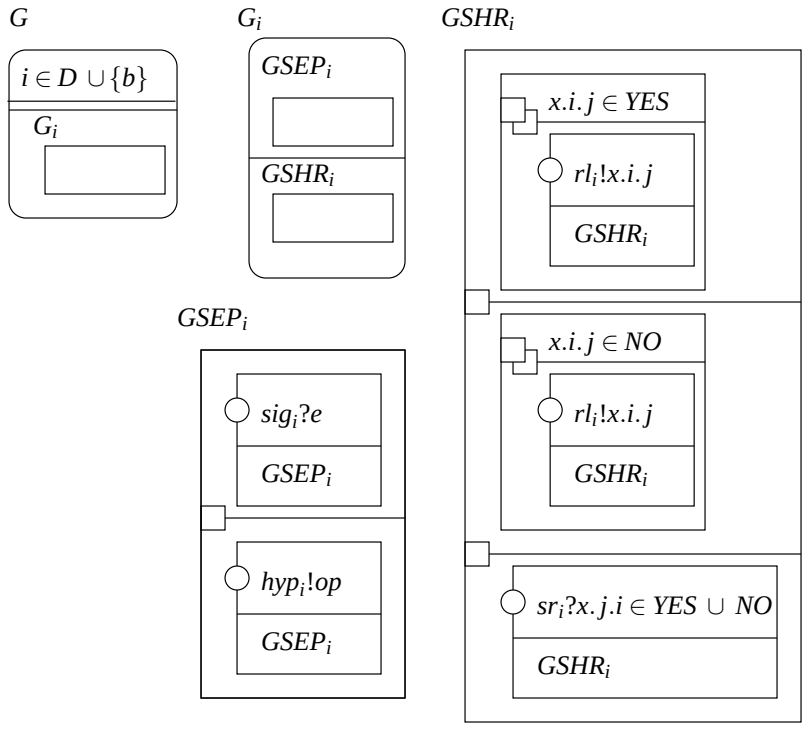


Figure 2: GSPML Diagrams of Guest Structures

sharing behavior. The alphabet of process $GSEP_i$ is disjoint from the alphabet of process

$GSHR_i$. The alphabet of a guest process G_i is F_i. Notice that the guest processes G_i do not have any events in common with each other; that is their alphabets F_i are pairwise disjoint and we could have connected them via alphabetized parallel as $\|_0^b\ (G_i, F_i)$ and have the same meaning.

Each guest process can choose values from $YES \cup NO$ to request sharing via the boundary controller. The boundary controller process makes sharing policy decisions and forwards approved data to the appropriate security domain, through the VMM.

2.4 Boundary Controller

The boundary controller process G_b models enforcement of the sharing policy. Figure 3 shows a GSPML diagram of the boundary controller sharing. Equation 4 shows the struc-

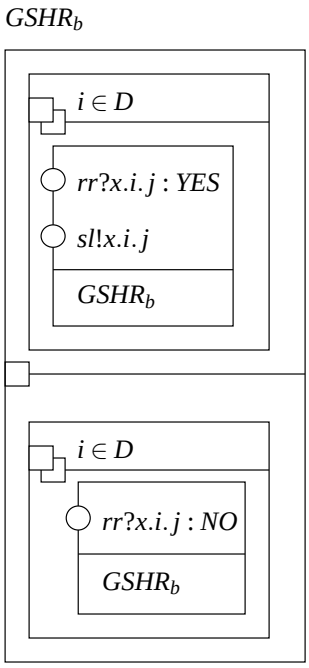


Figure 3: GSPML Diagram of Boundary Controller Sharing

ture of the boundary controller. The boundary controller is designed to honor requests in *YES* and discard the others.

$$G_b = GSEP_b \ \|\|\| \ GSHR_b \tag{4}$$

where

$$GSEP_b = hyp_b!op \rightarrow GSEP_b \ \Box \ sig_b?e \rightarrow GSEP_b$$

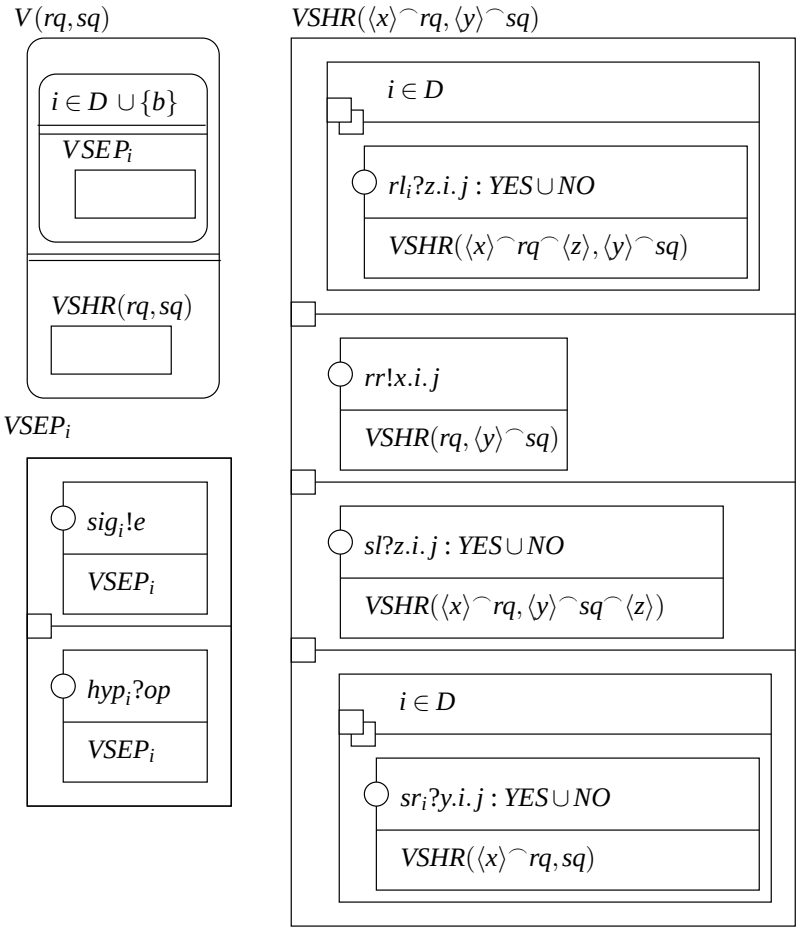


Figure 4: GSPML Diagrams of VMM Structures

and

$$GSHR_b =$$
$$\Box_{i \in D}\ rr?x.i.j : YES \rightarrow sl!x.i.j \rightarrow GSHR_b \,\Box$$
$$\Box_{i \in D}\ rr?x.i.j : NO \rightarrow GSHR_b$$

The alphabet of the boundary controller is referred to as F_b and the combined alphabets of all guests including the boundary controller is F. The boundary controller communicates with the other guests through the VMM.

2.5 VMM

The VMM process $V(rq, sq)$ defined on alphabet F not only responds to hypercall events and generates signals, it also connects sharing requests and responses via communication

buffers. Figure 4 shows GSPML diagrams of the VMM process structure. The top-level structure of the VMM process is

$$V(rq, sq) = (\ |||\ VSEP_i\ )\ |||\ VSHR(rq, sq) \tag{5}$$

The separation components $VSEP_i$ of the VMM are defined as

$$VSEP_i = hyp_i?op \rightarrow VSEP_i \,\Box\, sig_i!e \rightarrow VSEP_i$$

for all domains i in D and the boundary controller domain b.

Sharing via the VMM process is performed by a single process $VSHR(rq, sq)$ that connects sharing requests and responses via communication buffers rq (request queue) and sq (sharing queue). These communication buffers give the $VSHR(rq, sq)$ a parameterized and double recursive structure. Readers familiar with this kind of CSP construction may skip over the next three equations below to Equation 9. In Equation 9 and the three base cases preceding it, the variables $x, y,$ and z represent $x.i.j, y.i.k,$ and$z.i.\ell \in YES \cup NO$, respectively.

The initial base case is the VMM sharing process with no requests to pass and no data to share, Equation 6. It can only accept a single value to be queued to its request queue rq or sharing queue sq. Notice that use of the sl channel does not require an indexed choice over security domains D because channel sl only connects to the boundary controller G_b.

$$\begin{aligned} VSHR(\langle\rangle, \langle\rangle) = \\ &(\Box_{i \in D}\, rl_i?z : YES \cup NO \rightarrow VSHR(\langle z\rangle, \langle\rangle)) \\ &\Box\, sl?z : YES \cup NO \rightarrow VSHR(\langle\rangle, \langle z\rangle) \end{aligned} \tag{6}$$

The next base case, Equation 7, is a VMM sharing process with a non-empty, possibly singleton request queue $\langle x\rangle^\frown rq$ and no data to share (i.e. $sq = \langle\rangle$). It can accept values to enqueue or it can forward its single request to the boundary controller. Notice that use of the sl and rr channels do not require an indexed choice over security domains D because channels sl and rr only connect to the boundary controller G_b.

$$\begin{aligned} VSHR(\langle x\rangle^\frown rq, \langle\rangle) = \\ &(\Box_{i \in D}\, rl_i?z : YES \cup NO \rightarrow VSHR(\langle x\rangle^\frown rq^\frown \langle z\rangle, \langle\rangle)) \\ &\Box\, rr!x \rightarrow VSHR(rq, \langle\rangle) \\ &\Box\, sl?z : YES \cup NO \rightarrow VSHR(\langle x\rangle^\frown rq, \langle z\rangle) \end{aligned} \tag{7}$$

The final base case is a VMM sharing process with a non-empty, possibly singleton, sharing queue $\langle y\rangle^\frown sq$. It can forward shared value y to its destination guest, or it can enqueue

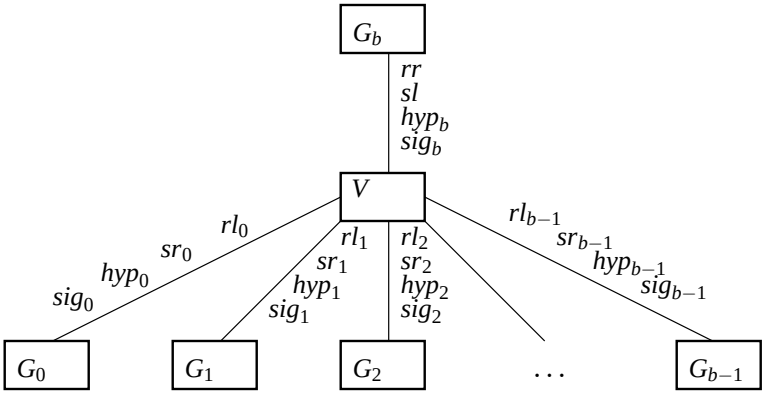


Figure 5: Communication Diagram

further requests or sharing.

$$
\begin{aligned}
&VSHR(\langle\rangle, \langle y\rangle^\frown sq) = \\
&\qquad (\Box_{i\in D}\, rl_i?z : YES \cup NO \rightarrow VSHR(\langle z\rangle, \langle y\rangle^\frown sq)) \\
&\qquad \Box\ sl?z : YES \cup NO \rightarrow VSHR(\langle\rangle, \langle y\rangle^\frown sq^\frown\langle z\rangle)\ \Box \\
&\qquad (\Box_{i\in D}\, sr_i!y \rightarrow VSHR(\langle\rangle, sq))
\end{aligned}
\tag{8}
$$

A VMM sharing process with both requests and data to share can interact with its guests over all possible process choices.

$$
\begin{aligned}
&VSHR(\langle x\rangle^\frown rq, \langle y\rangle^\frown sq) = \\
&\qquad (\Box_{i\in D}\, rl_i?z : YES \cup NO \rightarrow VSHR(\langle x\rangle^\frown rq^\frown\langle z\rangle, \langle y\rangle^\frown sq)) \\
&\qquad \Box\ rr!x \rightarrow VSHR(rq, \langle y\rangle^\frown sq) \\
&\qquad \Box\ sl?z : YES \cup NO \rightarrow VSHR(\langle x\rangle^\frown rq, \langle y\rangle^\frown sq^\frown\langle z\rangle)\ \Box \\
&\qquad (\Box_{i\in D}\, sr_i!y \rightarrow VSHR(\langle x\rangle^\frown rq, sq))
\end{aligned}
\tag{9}
$$

2.6 Complete Model

Given the definition of $V(rq, sq)$ in Equation 5 we can now define the complete model as the interface parallel combination of the guest and the VMM.

$$S = V(\langle\rangle, \langle\rangle) \underset{\mathrm{F}}{\|}_{\mathrm{F}} G \tag{10}$$

Figure 5 shows the communication pattern for the complete model. As an example, suppose guest process G_1 wants to pass data x to guest process G_2; it sends a "request" message, via channels rl_1 and rr, to the boundary controller G_b, through the VMM process $V(rq, sq)$. If the boundary controller G_b decides that passing data x to security domain 2 is allowed, it sends a "share" message to the destination guest G_2. This sharing would be modeled by the trace

$$\langle rl_1.x.1.2,\ rr.x.1.2,\ sl.x.1.2,\ sr_2.x.1.2\rangle$$

At this point it is important to recall that this model is a policy model and not a formal interface or implementation model. Specific details of hypercalls and signals are not essential to the MSL separation policy and the sharing policy does not need to include details of how the VMM flows information on behalf of the boundary controller.

3 Rationale

The formal model presented here is consistent and complete with respect to the MSL policy. We can see that it is consistent and complete because

- the complete model of Equation 10 is deadlock free,
- the model of Equation 10 satisfies the MSL separation policy on the events in SEP, and
- the complete model of Equation 10 satisfies the MSL sharing policy on the events in SHR.

It is easy to see that, as our model is defined, satisfaction of the MSL separation policy is not possible with respect to the events in $SHR \cup REQ$. Since the security class of each event is the highest domain that shares it, every boundary controller event has the highest classification. Even if the boundary controller only passes values in the allowable direction, there will be *High* events that influence the future behavior of guest processes at lower security classes. That is, event $sr_i?x.j.i$ from Equation 3 will or will not happen some time in the future, even though the domain controller's pass of value x conforms to the security lattice flow. This is intuitively reassuring because it hints at the necessity of dealing with the covert channels present in reliable interdomain sharing, as described by Kang, et al. [21–23]. We can show that there are no flows out of security domain i due to separation policy events $hyp_i.op, sig_i.e \in SEP_i$, for any security domain i. This is the value of the independence definition: we can show that the other guests are independent of the separation policy events SEP_i involving guest G_i.

The sharing policy can be verified using a *rank function approach* that models the propagation of facts over events in $SHR \cup REQ$. Intuitively, if the policy is enforced, then no fact will be output by a guest process contrary to the sharing policy's allowed flows.

In the following sections we sketch formal proofs of these three properties. Mechanical proofs are left to future work.

3.1 Deadlock Freedom

The model's failures equivalence definition of deadlock freedom is adapted from Roscoe [11] and Schneider [18] as

$$(tr, \Sigma) \notin failures(S) \tag{11}$$

Intuitively, after any possible trace tr of the model of Equation 10, there must be some event that the composite process S cannot refuse.

For our model to be deadlock free, it must be divergence-free. We show this by noticing that the processes defined in Equations 3, 4, and 5 have guarded recursion and contain only operators that do not introduce diversion. The interleaving of the guest processes $|||\, G_i$ introduces no divergence because the events of the guests are mutually disjoint. By the same reasoning, the complete model of Equation 10 combines two divergence-free processes using alphabetized parallel, so the complete model is divergence-free.

Given the divergence-free system S of Equation 10, we can show that it is also deadlock free. First, we notice that the following conditions hold

A. None of the component processes terminate.

B. The communications of S in Equation 10 are static and the communications of the components are entirely within their corresponding alphabets. There is no ambiguity as to which processes participate in any event of S.

C. The communications of S are *triple disjoint* [11]. That is, no event requires synchronization by more than two component processes.

D. There is no use of renaming or hiding operators that potentially introduce hidden events.

E. We assume that each component is *busy*, that is the component itself is deadlock-free.

where component process means any process in either the guests G or the VMM V.

We argue that the design of Equation 10 is deadlock-free essentially because there is no *strong conflict* [11] between any pair of communicating components; i.e. there are no pairs of components where each has an ungranted request from the other. Because the communication diagram of S is a tree (see Figure 5), process S cannot have a deadlock that involves a proper cycle of components,i.e more than two components. Given no strong conflict, there cannot be a cycle that involves only two components.

3.2 Separation

Our proof that the model of Equation 10 satisfies the MSL separation policy over events in SEP_i applies a corollary of Roscoe [17][2]: *Process S is deterministic and* separable over alphabets H and L *if and only if the processes $\mathcal{L}_H(S)$ and $\mathcal{L}_L(S)$ are both deterministic.*

[2] Corollary 12.1.3

To be separable, a process P must first be divergence-free and nonterminating, i.e. no $\checkmark$ events. Additionally, process P must also be equivalent to $P = P_H \;|||\; P_L$, with its alphabet partitioned into H and L.

Our proof strategy will be to show that the process S of Equation 10 is deterministic and separable, by its construction.

Instead of restricting all flows from *High* to *Low*, we only restrict flows caused by separation policy events $hyp_i.op$ and $sig_i.e$, for each guest i. Essentially, this rules all events in $SHR \cup REQ$ to be *Low* events, for separation purposes. Let disjoint event sets $H = SEP_i = \{hyp_i.op, sig_i.e\}$, for some security domain i, and $L = F \setminus H$ be the *High* and *Low* events. Then we have the *lazy abstraction* of S

$$\mathcal{L}_H(S) = (S \;{}_F\!\parallel_H\; CHAOS_H) \setminus H \tag{12}$$

Given the lazy abstraction of S we can formulate the separation policy as

$$\mathcal{L}_H(S)\ \mathsf{det} \tag{13}$$

That is, the lazy abstraction must be deterministic, as explained in Equation 2.

The separation policy of Equation 12, that is the lazy abstraction $\mathcal{L}_H(S)$, is not necessarily deterministic because $CHAOS_H = STOP_H \sqcap RUN_H$ is nondeterministic. To show that our separation policy is satisfied, we apply the corollary by showing that process S of Equation 10 is both deterministic and *separable*.

We can say that process S is deterministic by its construction. The guests defined by Equations 3 and 4 are deterministic because they use constructive recursion[3] defined on an external choice over component processes with disjoint *initials*. Since the *initials* of the external choice are disjoint, there is no ambiguity over which component process has been chosen. In a similar way, the use of the interleaving operator $|||$ which has the potential to introduce nondeterminism, does not because the events of each guest are disjoint. So the interleaved combination of guests $|||\, G_i$ is deterministic. The VMM process $V(rq, sq)$ of Equation 5 is deterministic by 1) its use of constructive recursion over external choice with disjoint *initials* and 2) disjoint event alphabets that avoid nondeterminism introduced by interleaving. The alphabetized parallel operator joining the guests and the VMM does not introduce nondeterminism, so the complete model of Equation 10 is deterministic, that is S det.

The second part of our argument is to show that process S is separable over alphabets H and L. By definition, event alphabets H and L given at the beginning of this section are partitioned. We need to show that process S of Equation 10 is equivalent to a process

$$S = (\, VSEP_i \;{}_{\mathrm{H}}\!\parallel_{\mathrm{H}}\; GSEP_i) \;|||\; (V' \;{}_{\mathrm{L}}\!\parallel_{\mathrm{L}}\; G')$$

where processes V' and G' are the remainder of G and V after the H components $VSEP_i$ and $GSEP_i$ are factored out. Our argument makes use of two algebraic properties of CSP.

[3] Intuitively, the recursion does not involve use of the hiding operator. For a more formal discussion, see Chapters 3 and 9 of Roscoe [17] and also page 106 of Chapter 4.

The first property is the fact that $P \;|||\; Q = P \;{}_{\mathrm{X}}\|_{\mathrm{Y}}\; Q$, if the alphabets X and Y are disjoint. We refer to this first property as *disjoint alphabetized parallel equivalence*. The second property is the associativity of alphabetized parallel, that is

$$(P \;{}_{\mathrm{X}}\|_{\mathrm{Y}}\; Q) \;{}_{X \cup Y}\|_{\mathrm{Z}}\; R = P \;{}_{\mathrm{X}}\|_{Y \cup Z}\; (Q \;{}_{\mathrm{Y}}\|_{\mathrm{Z}}\; R) \tag{14}$$

We begin by partitioning the alphabets into H and L, so that Equation 10 becomes

$$V \;{}_{H \cup L}\|_{H \cup L}\; G$$

Next we apply the commutativity of the interleaving operator to G, to move the selected H component $GSEP_i$ to the left. Applying disjoint alphabetized parallel equivalence to G, we get

$$V \;{}_{H \cup L}\|_{H \cup L}\; (GSEP_i \;{}_{\mathrm{H}}\|_{\mathrm{L}}\; G') \tag{15}$$

We now apply the associativity of alphabetized parallel. Interpreting V as P, $GSEP_i$ as Q, and G' as R, from the RHS of Equation 14, we define the interfaces from left to right as $H \cup L = X, H \cup L = Y \cup Z, H = Y$, and $L = Z$. With this interpretation of processes and interfaces, associativity applied to Equation 15 gives us

$$(V \;{}_{H \cup L}\|_{\mathrm{H}}\; GSEP_i) \;{}_{H \cup L \cup H}\|_{L}\; G' \tag{16}$$

Next we move $VSEP_i$, the H component of V, to the right, using commutativity of interleaving. Applying disjoint alphabetized parallel equivalence to V, we get

$$((V' \;{}_{L}\|_{H}\; VSEP_i) \;{}_{H \cup L}\|_{\mathrm{H}}\; GSEP_i) \;{}_{H \cup L}\|_{L}\; G' \tag{17}$$

Now we can apply associativity of alphabetized parallel to Equation 17. From the LHS of Equation 14 we interpret $V' = P$, $VSEP_i = Q$, and $GSEP_i = R$, defining the interfaces from left to right as $L = X, H = Y, H \cup L = Y \cup X$, and $H = Z$. With this interpretation of processes and interfaces, associativity gives us

$$(V' \;{}_{L}\|_{H \cup H}\; (VSEP_i \;{}_{H}\|_{H}\; GSEP_i)) \;{}_{H \cup L}\|_{L}\; G' \tag{18}$$

Applying commutativity we can move $(VSEP_i \;{}_{H}\|_{H}\; GSEP_i)$, which plays the role of P_H in our argument, to the left

$$((VSEP_i \;{}_{H}\|_{H}\; GSEP_i) \;{}_{H}\|_{L}\; V') \;{}_{H \cup L}\|_{L}\; G' \tag{19}$$

A final application of associativity of alphabetized parallel, to Equation 19, applies the LHS of Equation 14 as $(VSEP_i \;{}_{H}\|_{H}\; GSEP_i) = P, V' = Q, G' = R$. Defining the interfaces from left to right as $H = X, L = Y, H \cup L = X \cup Y$, and $L = Z$ gives us

$$(VSEP_i \;{}_{H}\|_{H}\; GSEP_i) \;{}_{H}\|_{L \cup L}\; (V' \;{}_{L}\|_{L}\; G') \tag{20}$$

To complete the argument, we apply disjoint alphabetized parallel equivalence

$$(VSEP_i \underset{H}{\|}_{H} GSEP_i) \;|||\; (V' \underset{L}{\|}_{L} G') \tag{21}$$

Since we now have factored process S into two interleaved components over disjoint alphabets (as in $P = P_H \;|||\; P_L$), we have S as separable. Given our previous reasoning that S by itself is deterministic, we can now apply Roscoe's corollary to show that the lazy abstraction of S is deterministic. Since $\mathcal{L}_H(S)$ det, our separation policy holds for Equation 10.

3.3 Sharing

To show that the sharing is secure, we must demonstrate that a sharing policy event $sr_j.x.i.j \in SHR$, where $x.i.j \in NO$, does not appear in any trace of the system. Sharing request events $rl_i.x.i.j \in REQ$, where $x.i.j \in NO$, do appear in traces(G_i) of a guest; guests other than the boundary controller request all possible forms of sharing. However, we need to prove that Equation 10 defines sharing such that only $x.i.j \in YES$ values are passed from any guest G_i to any other guest G_j.

Our approach is to use the rank function approach first reported by Schneider [24]. The rank function approach assigns integers to the events and values in a security protocol, with a rank function, and then shows that only events with positive rank can occur in a run of the protocol. In our secure sharing argument, we consider the VMM and its guests to be principals in a security protocol and apply a rank function. We then show that only values of positive rank are shared. Our argument differs from the usual rank function approach because non-positive values do appear in traces(S). Instead of showing that no principal receives a non-positive value, we show that the essential principal, the guard, never sends a non-positive value, even though some non-positive values are submitted to it. The simple construction of the guard makes it easy to establish this. Conventional security protocols are more complex and require the complete approach shown by Schneider [24].

The rank function approach is well-suited to what we have to show. At the beginning of this section we pointed out that events in $REQ \cup SHR$ can cause non-determinism, so secure sharing is not defined according to the independence of G_j's view of the rest of S. Instead, sharing is secure if none of the values that appear on G_j's sharing channel sr_j are unauthorized, i.e. in the set NO.

For our proof we use the rank function ρ shown below as Equation 22.

$$\rho(a) = \begin{cases} 0 & \text{if } a = sr_j.x.i.j,\ x.i.j \in NO \\ 0 & \text{if } a = sl.x.i.j,\ x.i.j \in NO \\ 1 & \text{otherwise} \end{cases} \tag{22}$$

Our argument does not need to address model components $GSEP_i$ or $VSEP_i$ because they do not contain events of rank 0.

We begin by noticing that, by definition, values $x.i.j \in NO$ do not appear on channel sl. That is, the boundary controller process satisfies the property G_b sat no $(\{sl.x.i.j\} \subseteq NO)$;

all of the values associated with channel sl must be in YES, by the design of $GSHR_b$ shown in Equation 4.

Given G_b sat no $(\{sl.x.i.j\} \subseteq NO)$ we can now say that $G_b \underset{F_b}{\|}_{F} V(rq, sq)$ sat no $(\{sl.x.i.j\} \subseteq NO)$ as well, because the alphabetized parallel operator constrains what the VMM process $V(rq, sq)$ can do. So we are now able to say that no event of non-positive rank happens because of model component $G_b \underset{F_b}{\|}_{F} V(rq, sq)$. This component does not generate such events on channel sl and it is the only place in the whole model that they could be generated.

Given the absence of an event $sl.x.i.j \in NO$ in process $G_b \underset{F_b}{\|}_{F} V(rq, sq)$ we can now say that sharing queue sq never contains a value $x.i.j \in NO$. By its construction, the VMM process can never generate an event $sr_j.x.i.j,\ x.i.j \in NO$ because the VMM process' sharing queue never contains the value needed to generate such an event. So model component $V(rq, sq)$ sat no $(\{sr_j.x.i.j\} \subseteq NO)$. Given this, we can say that the alphabetized parallel combination of the VMM and any guest process G_j also does not have an $sr_j.x.i.j,\ x.i.j \in NO$ event in any of its traces. That is $G_j \underset{F_i}{\|}_{F} V(rq, sq)$ sat no $(\{sr_j.x.i.j\} \subseteq NO)$. Since this is the only component that could generate such an event, we can say that no event of non-positive rank happens because of component $G_j \underset{F_i}{\|}_{F} V(rq, sq)$. Having ruled out the only two events of non-positive rank, we can say that the model of Equation 10 maintains positive rank with respect to the rank function of Equation 22.

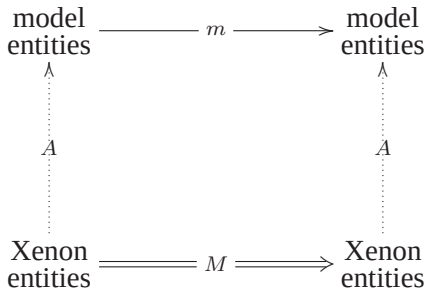
4 Semiformal Correspondence Demonstration

To apply this model to Xenon, we construct a semiformal correspondence demonstration. This correspondence shows how the model represents Xenon's actual policy enforcement. Constructing and applying the correspondence demonstration helps us to identify interface and design problems with the VMM that might not be apparent from the source code alone. For all but the most casual software security, some form of correspondence demonstration is essential because security is not preserved by refinement [9, 25].

The concept of a correspondence demonstration is based on the commutative diagram principle for verifying data abstraction [26–28]. [4] A commutative diagram captures the relationship between concrete software and an abstract model. Within the model (or the software), we are interested in the relationships between the entities defined by the model (or the software), captured in the diagram as a mapping m between model (or M for the software) objects. The model is shown to correspond to the software by a *representation function* A that maps concrete software entities to abstract model entities. Equation 23 shows a commutative diagram for correspondence to the Xenon formal model. An abstract model properly corresponds to its target concrete software if the diagram *commutes*. A

[4] We recall this for the reader because this knowledge seems to have been lost in some parts of the security community.

diagram commutes if and only if, starting at the lower left corner, moving in both directions to the upper right corner gives the same result, wherever the abstraction is defined. For Equation 23 this is $m \circ A = A \circ M$.



(23)

For security, it is important that the representation function A should be a total function that maps all of the entities in the Xenon interface into entities in the formal security policy model. Otherwise, a flaw introduced by some unmapped software entity will not be caught by the policy-to-code modeling. For a similar reason, we prefer not to have don't care conditions in our modeling that would lead $m \circ A \subseteq A \circ M$.

The commutative diagram principle does not have to be applied formally to benefit software security. Semiformal mappings constructed from natural language rules and tables can emulate a formal representation mapping.

Our semiformal correspondence demonstration is based on mapping the Xenon VMM interface to pertinent characteristics of CSP events and processes of the formal model. We view the hypercalls, additional traps that could happen, interrupts, individual instructions, and high-level language statements attempted by each guest as "Xenon events". We show correspondence by mapping these "Xenon events" to CSP events in the formal model.

The Xenon interface has entities besides events and we will show how these other entities map to the formal model, after we understand the basic map. Our correspondence demonstration should be considered as a separate model, being neither the formal security policy model nor the abstract VMM described by the semiformal specification of the VMM interface.

4.1 Basic Mapping

The basic semiformal correspondence demonstration maps each "Xenon event" into an event in the formal model. Xenon events include hypercalls, interrupts, traps, unprivileged instructions, and privileged instructions. Privileged instructions include not only the instructions that cause a guest OS execution to exit to the VMM but also the virtual machine control instructions like VMX and VME[5]. Unprivileged instructions are also mapped. The basic map is given as Table 1.

Although the basic map given in Table 1 is sufficient to show that the formal policy model corresponds to the VMM interface, it may not be clear how this map is to apply to

[5]Xenon is currently targeted at 64-bit x86 architectures only.

Xenon VMM Interface	Formal Policy Model
guest OS	G_i
domain	G_i
hypercall	$hyp_i.op$
unprivileged instruction	$hyp_i.op$
privileged instruction	$hyp_i.op$
trap	$hyp_i.op$
interrupt	$sig_i.e$
received event channel message	$sig_i.e$

Table 1: Basic Xenon Interface to Formal Policy Model Map

all details of the VMM interface. Some further interpretation is necessary.

4.2 Externally Visible Data

The VMM interface provides data structures that are visible outside the VMM. These data structures include virtual memory, virtual machine control structures, device driver front ends, and structures like event channels that are presented as part of the Xen interface. In principle, all effects of these data structures can be mapped on the basis of reads and writes to the applicable memory, i.e. as "Xenon events". Each read or write (or high-level language operation on data) can be viewed as one or more "Xenon events" that map directly to a CSP event. The particular events that happen define the state of each piece of data. In some cases, this direct application of "Xenon events" will be the simplest way to perform the mapping of data to CSP model events.

In practice this direct application can be too complex. In those situations we represent data in the Xenon interface by CSP processes. Using processes to model data structures is a conventional CSP technique. Techniques for modeling shared data as processes are well-defined and widely understood by the CSP community [17, 18].

To apply this technique, we first model memory, devices, and data structures in the Xenon interface specification as CSP processes. We define one process for each instance of a data structure. With the data transformed into CSP processes, we can then map the CSP events associated with those processes into the CSP events of the formal model. Complexity is reduced because we can interchangeably refer to the interface data structures as though they were processes or as the actual data structures.

Each of these "memory" processes defined by the Xenon interface is represented using well-established CSP practice via "set" and "get" events that model reading and writing of the corresponding data structure. Race conditions and similar issues then emerge as trace-level characteristics of these processes, since CSP offers no built-in guarantees against race conditions or similar issues. Modeling externally visible data as CSP processes does not magically hide any of the potential difficulties that can arise with the real data structures

they represent.

So it is possible to simplify the mapping to the formal model by thinking about each data structure by its name, but treating the data structure as though it were a process. The fact that some CSP "processes" in this conceptual VMM interface are data structures is irrelevant to the separation policy.

4.3 Request-Response and Transients

The semiformal correspondence demonstration introduces additional concepts that do not appear in the formal model. The correspondence demonstration is a refinement of the formal model, though the refinement is no longer mathematically well-defined. The first refinement is that request and response (send and receive) *may* no longer map from the Xenon interface as a single event. It is desirable to retain the single-event representation in the correspondence demonstration, as much as possible. Request and response will have to be separated in those cases where mapping from a single Xenon interface event does not preserve the CSP properties critical to the MSL information flow policy.

The actual VMM interface behavior will also include transient events such as locking and non-atomic changes to memory. In the formal model, transients could be modeled in CSP by using hidden τ events. Doing this would significantly increase the complexity of the formal model and introduce non-determinism as a general property of the security policy. If the formal model were non-deterministic by its construction, then we could not use determinism as the fundamental definition of security. So the formal model is deterministic, without transient events. The correspondence demonstration introduces transient events as semiformal concepts. The correspondence demonstration then has to show that the transient events in the Xenon interface satisfy the essential properties of CSP events necessary for MSL information flow policy enforcement, i.e. they cannot be used to signal from one domain to another.

4.4 Nondeterministic Interrupts

The formal model does not address the nondeterminism of interrupts: events that come from the environment rather than from the VMM or one of the guests. In the model, interrupts are represented by signal events arising deterministically within the VMM. CSP can model nondeterministic interrupts accurately, via a special interrupt operator $\triangle$. The significant issue here is that the accurate CSP interrupt operator $\triangle$ introduces nondeterminism to the security policy, while the model's definition of security requires determinism of all events. (Accurate modeling would also significantly increase the complexity of the model and distract us from security policy concepts.)

In the Xenon interface, interrupts will come from outside the system, in a nondeterministic way. The issue then becomes one of showing that untrusted guests cannot cause or prevent interrupts from the environment to another guest. This will justify the mapping of nondeterministic Xenon interrupts into the deterministic signal events of the formal policy model.

4.5 Subjects and Objects

The model's separation policy given by Equation 12 defines security as determinism in the lazy abstraction $\mathcal{L}_H(S)$. Our correspondence demonstration may need to map the more familiar subjects and objects into this definition, to satisfy some Common Criteria requirements. Xenon domains in the interface are security policy subjects and Xenon resources (e.g. memory or devices) in the Xenon interface are security policy objects. So our mapping can show how Xenon subjects and objects fit into the formal policy model.

In the correspondence demonstration, Xenon domains then become guest processes of the model, as shown in Equations 3 and 4. The data structures that implement domains in Xenon will have correspondence processes for their memory, devices, CPU-defined structures, and Xenon-defined abstractions associated with a Xenon domain. Xenon's domain0 is part of the VMM, in the correspondence demonstration. That is, domain0 is not subject to the restrictions of either the separation policy or the sharing policy.

4.6 Least Privilege

Levin, Irvine, and Nguyen have published a *least privilege separation model* [29] for static separation kernels. This model defines fine-grained access control over individual resources exported to a domain. It also envisions fine-grained exportation of subjects; that is, the least privilege model defines more than one subject per domain (using Xenon terminology). The least privilege model is a very practical model that exemplifies the inevitable intertwining of formal modeling and ultimate implementation. Least privilege separation anticipates underlying hardware with segment registers. Segment registers are superior for enforcing separation via hardware, for two reasons: 1) segment registers are easily associated with specific hardware processes, and 2) detailed security attributes, i.e. a large number of bits per process, can be amortized over many bytes of memory. Implementing a least privilege policy with page-based memory management would be awkward and inefficient.

Xenon will be implemented on 64-bit x86 hardware. This hardware disables the segment registers, so it will not be possible for Xenon or any other VMM to enforce a least privilege model on 64-bit x86 hardware; at least not with any kind of arguable simplicity. On the other hand, Xenon's multiple single levels (MSL) policy provides for a different kind of least privilege, because it limits the interactions between guests to a single reliable upward replication. We can also limit the privileges of the boundary controller G_b by dividing it into a family of single-interaction (i.e. one direction between one pair of security domains) boundary controllers G_{ij} that control sharing from security domain i to j. While this would have no impact on the formal proof of security[6], the model would become cluttered, since this would require $n(n-1)$ boundary controller processes in the model. In practice, we only need single-interaction boundary controllers where the defined security policy calls for interaction.

[6] We can use the VMM process V as the root of the necessary tree-structured communication diagram.

4.7 Using the Model and Correspondence Demonstration

With these interpretations of the basic mapping established, we can now restate the separation policy of Equation 12 in the following semiformal manner.

> If the history of the VMM and all its domains determine that a Xenon event of a guest in a low domain is possible (or impossible) then a guest in a high domain cannot cause the same Xenon event to be impossible (resp., possible).

Recalling that reads or writes to Xenon domains will be "Xenon events" of the corresponding processes, we see that a key implication of this policy is that no domain should be able to read from or write to parts of any other domain.

This key implication is not the whole application of this model. Each interrupt, trap, hypercall, and instruction is a distinct semiformal concept that must be validated against the mapping. A high domain should not be able to modulate the possibility or impossibility of any "Xenon event" in a low domain, if separation applies.

5 Summary

The Xenon formal security policy model combines both separation and sharing information flow policies in one model. Previous formal security policy models only addressed separation; sharing has been either left out or assigned to trusted subjects that are not modeled.

The composability of CSP and its general suitability for modeling event-based noninterference policies makes it possible to construct a combined model and a relatively simple one. The structure of the policy model is intuitively close to the structure of an implementation, though it imposes no structure on any valid implementation. We have shown how easy and natural it is to construct a semiformal mapping from the more complex world of the Xenon interface to our model.

Our future plans for the model include 1) using it to design Xenon and 2) constructing mechanical proofs either directly using a CSP-based tool like FDR or by embedding CSP into another formalism like ACL2.

References

[1] P. Barham, B. Dragovic, K. Fraiser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP-19)*, Bolton Landing, New York, USA, October 2003.

[2] R. Sailer, T. Jaeger, E. Valdez, R. Cáceres, R. Perez, S. Berger, J. Griffin, and L. van Doorn. Building a MAC-Based security architecture for the Xen open-source hypervisor. In *Proc. 21st Annual Computer Security Applications Conference*, Tucson, Arizona, US, December 2005.

[3] P. Karger. Multi-level security requirements for hypervisors. In *21st Annual Computer Security Applications Conference (ACSAC)*, Tucson, AZ, US, December 2005.

[4] The Common Criteria Project Sponsoring Organizations. *Common Criteria for Information Technology Security Evaluation*, v. 3.1, rev. 1 edition, September 2006. also referred to as ISO 15408.

[5] J. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Research in Security and Privacy*, Oakland, California, US, April 1982.

[6] D. Denning. *Secure Information Flow in Computer Systems*. PhD thesis, Purdue University, West Lafayette, Indiana, US, 1975.

[7] D. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5), May 1976.

[8] D. Bell and L. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306,MTR-2997 rev. 1, MITRE, 1976.

[9] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Research in Security and Privacy*, Oakland, California, US, May 1994.

[10] A. Roscoe, J. Woodcock, and L. Wulf. Non-interference through nondeterminism. In *Proc. ESORICS*, Brighton, UK, November 1994.

[11] A. Roscoe. CSP and determinism in security modelling. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, US, May 1995.

[12] P. Ryan and S. Schneider. Process algebra and non-interference. In *Proc. 12th IEEE Computer Security Foundations Workshop*, Mordano, IT, June 1999.

[13] E. Kleiner and T. Newcomb. On the decidability of the safety problem for access control policies. In *Sixth International Workshop on Automated Verification of Critical Systems (AVoCS)*, Nancy, FR, September 2006.

[14] W. Ruzzo, M. Harrison, and J. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[15] D. Greeve, M. Wilding, and W. M. Vanfleet. A separation kernel formal security policy. In *Proc. ACL2 Workshop*, Boulder, Colorado, US, July 2003.

[16] J. Alves-Foss and C. Taylor. An analysis of the gwv security policy. In *Proc. ACL2 Workshop*, Austin, Texas, US, November 2004.

[17] A. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International, 1997.

[18] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. John Wiley and Sons, Ltd., 2000.

[19] J. McDermott and G. Allwein. A formalism for visual security protocol modeling. *Journal of Visual Languages and Computing*, accepted October 2006.

[20] J. McDermott. A formal syntax and semantics for the GSPML language. Technical report, U.S. Naval Research Laboratory, 2005.

[21] M. Kang, I. Moskowitz, and S. Chincheck. The pump: a decade of covert fun. In *Proc. Annual Computer Security Applcations Conference*, Tucson, Arizona, US, December 2005.

[22] M. Kang and I. Moskowitz. A pump for rapid, reliable, secure communications. In *Proc. ACM Conf. on Computer and Communications Security*, Fairfax, Virginia, US, November 1993.

[23] M. Kang, I. Moskowitz, and D. Lee. A network pump. *IEEE Trans. on Software Engineering*, 22(5), 1996.

[24] S. Schneider. Verifying the correctness of authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9):741–758, September 1998.

[25] D. McCullough. Specifications for multi-level security and a hook-up property. In *IEEE Symposium on Security and Privacy*, Oakland, California, US, May 1987.

[26] C. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[27] W. Wulf, R. London, and M. Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Trans. on Software Engineering*, SE-2:253–265, 1976.

[28] J. Gannon, R. Hamlet, and H. Mills. Theory of modules. *IEEE Trans. on Software Engineering*, SE-13(7):820–829, July 1987.

[29] T. Levin, C. Irvine, and T. Nguyen. A least privilege model for static separation kernels. Technical Report NPS-CS-05-003, U.S. Naval Postgraduate School, October 2004.